H-A

03CO ~~0420~~

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

Applicant: Halvarsson, *et al.*

~~Serial No.: 09/738,720~~

Filed: December 15, 2000

For: DEVICE FOR DATASTREAM DECODING

Group Art Unit: Not As Yet Assigned

Examiner: Not As Yet Assigned

Attorney Docket: 6515-53621

CERTIFICATE OF MAILING/TRANSMISSION (37 C.F.R. § 1.8A)

I hereby certify that this correspondence is, on the date shown below, being:
(X) deposited with the United States Postal Service with sufficient postage as first class mail in an envelope addressed to: Assistant Commissioner for Patents, Washington, D.C. 20231.
( ) transmitted by facsimile to the Patent and Trademark Office.

Date: May 2, 2001

Sharon Jones

TRANSMITTAL LETTER OF PRIORITY DOCUMENT

Assistant Commissioner for Patents
Washington, D.C. 20231

Sir:

Enclosed herewith is a certified copy of priority document SE 9904685-6 for the above-identified application.

Respectfully submitted,

Dated: May 2, 2001

Donald L. Bartels
Registration No. 28,282

COUDERT BROTHERS
4 Embarcadero Center, Suite 3300
San Francisco, CA 94111
Telephone: (415) 986-1300
Telefax: (415) 986-0320

SFO 4038997v1

PRV

PATENT- OCH REGISTRERINGSVERKET
Patentavdelningen

Intyg
Certificate

Härmed intygas att bifogade kopior överensstämmer med de handlingar som ursprungligen ingivits till Patent- och registreringsverket i nedannämnda ansökan.

This is to certify that the annexed is a true copy of the documents as originally filed with the Patent- and Registration Office in connection with the following patent application.

(71)	*Sökande* *Applicant (s)*	*SwitchCore AB, Lund SE*
(21)	*Patentansökningsnummer* *Patent application number*	*9904685-6*
(86)	*Ingivningsdatum* *Date of filing*	*1999-12-17*

Stockholm, 2000-12-19

För Patent- och registreringsverket
For the Patent- and Registration Office

A. Södervall
Anita Södervall

Avgift
Fee 170:-

CERTIFIED COPY OF PRIORITY DOCUMENT

A PROGRAMMABLE PACKET DECODER

Field of invention

A packet decoder assists a packet switch in identifying different types of packets from a data stream. Different types of packets have various lengths and headers located at different positions in different layers. The headers contain information needed by the switch, especially addresses. Also the data packets should be checked for errors, e.g. length errors and checksum errors. The packet decoder detects the types of packets and extracts pertinent information and inserts the result in the data stream. The switch also receives control signals on a parallel control line from the packet decoder.

Prior art

A conventional solution is to use delay lines on which masks are applied to filter out the required information. A fixed packet decoder has only one mask. It has also been suggested to provide a packet decoder with several masks which can be selected to adapt the decoder for different types of packets.

The invention

The present invention does not use masks. Instead the decoder is programmed to make comparisons and the various operations by means of a program memory and a compare processor.

The program memory has two parts. The first part of the program controls the detection of different types of packets and layers and also starts the proper saving procedure of the extracted information. This is the compare instruction memory.

The other part of the program memory contains sub-routines for controlling the operations on the information and saving of parts of the bit stream. This is the save instruction memory.

The packets arrive in a byte stream on a delay line, which is a shift register. In each clock cycle the byte stream is moved forward one step. A mux control unit detects the start of a packet and also keeps track of every byte as the packet is passing through the delay line.

The compare processor receives instructions from the compare instruction memory. Since the compare processor is double-ported it may receive two instructions per clock cycle. Each instruction may fetch the correct byte or bytes from the delay line by means of the mux control. Depending on the instruction, the compare processor orders a save engine to perform two different save operations.

In a bit save operation, the save instruction memory commands the bit save unit to perform one of three types of operations: a checksum control, a bit save, or a length error control. These are various check operations resulting in setting of flags in a result field.

In case of a stream save, the save instruction memory commands the stream save unit to extract bytes to be saved in an option field. The stream save unit inserts the option field as well as the result field in the byte stream outgoing to the switch. While the result field is inserted, the original byte stream is delayed in a 3 byte delay line. Also, control signals are sent on a parallel line to the switch.

The packet decoder is programmed by means of higher level languages to be able to take care of different types of packets. The program memory may easily be expanded, if necessary.

The invention is described in detail in the enclosed master thesis.

CLAIMS

1. A packet decoder detecting types of packets and extracting pertinent information from a data stream and inserting the result in the data stream as well as transmitting control signals on a parallel control line to a switch, wherein the packet decoder is programmed to make comparisons and the various operations by means of a program memory and a compare processor.

2. A packet decoder according to claim 1, wherein the program memory has two parts, the first part of the program controlling the detection of different types of packets and layers and also starting the proper saving procedure of the extracted information, the other part of the program memory containing sub-routines for controlling the operations on the information and saving of parts of the bit stream.

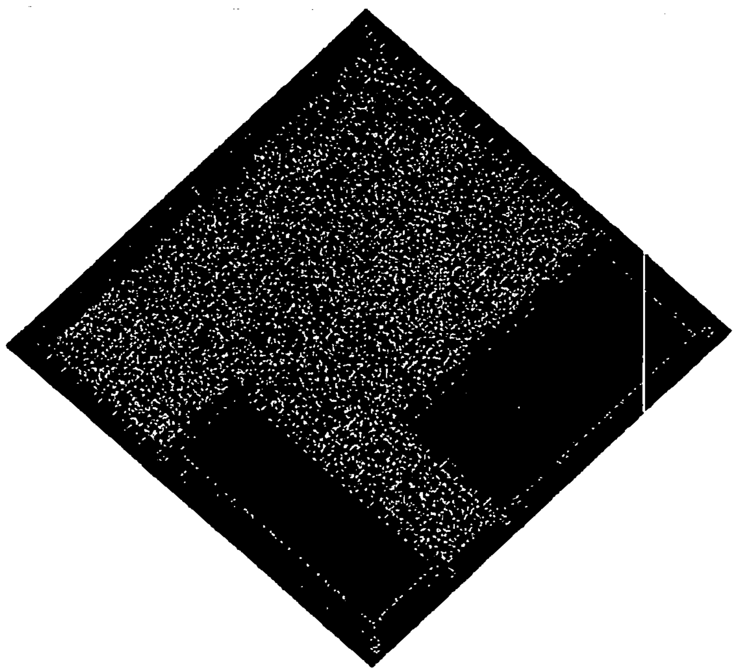
ABSTRACT

The invention relates to packet decoder detecting types of packets and extracting pertinent information from a data stream. The packet decoder inserts the result in the ~~data stream and transmits control signals on a parallel control line to a switch.~~ According to the invention, the packet decoder is programmed to make comparisons and the various operations by means of a program memory and a compare processor. The program memory has two parts, the first part of the program controlling the detection of different types of packets and layers and also starting the proper saving procedure of the extracted information. The other part of the program memory contains sub-routines for controlling the operations on the information and saving of parts of the bit stream.

Tintin
a programmable packet decoder
Master thesis by:

Ingemar Hammarstrom, d95ih
Stig Halvarsson, e94sh

December 15, 1999



1 Abstract

This paper describes the development of a programmable packet decoder called Tintin. Tintin is primarily developed for ethernet traffic in gigabit speed, but support other packet based traffic as well.

The input to Tintin is a byte wide data stream to be decoded. The output is the same data stream with two additional fields, the result field and the option field, describing each packet. The contents of these fields are specified with Tintins specialized assembly language.

Contents

CONTENTS *CONTENTS*

2 Preface

A study of this topic started during the course VLSI architecture, held at Lund Institute of Technology, Sweden, winter 98/99. The work lead to this master thesis. However, the design presented in this paper have been totaly rewritten.

3 Acknowledgement

The work was performed at SwitchCore AB, where the supervisor was Kenny Ranerup, *kenny.ranerup@switchcore.com*. Examiners were Peter Nilsson, *peter.nilsson@tde.lth.se*, and Viktor Owall, *viktor.owall@tde.lth.se*, both associate professors at the department of Applied Electronics, Lund Institute of Technology, Sweden.

Besides the examiners and supervisor we would like to thank the staff at Switchcore who helped us. A special thank to Lars-Olof Svensson for his help with the Apollo tools and Thomas Johansson for the pictures.

4 Ethernet networking

There is much use of common network phrases in this paper. For those who is not familiar with these, here is a basic network/ethernet description.

4.1 Protocol suites

To make computer network efficient, equipment from different vendors must be able to work together. To achieve this, a architecture standard for communication must exist. Here is some examples of such standards.

4.1.1 The OSI model

For many years there were no good standard covering the whole communication aspect. In 1977 ISO established a group to develop such a standard. 1984 the ISO group suggested a layer model, called the Open Systems Interconnection (OSI) model. It consisted of seven layers, all shown in figure 1.

Application
Presentation
Session
Transport
Network
Data link
Physical

Figure 1: The OSI environment

A layer describes the kind of information exchanged on that level. The higher layer is transparent to lower layer and a layer on one level do not know that lower levels exists, e.g. the transport layer thinks it communicates with another transport layers. In computer communication the rules of a layer is implemented in a protocol. There might be many kinds of protocols on one layer, usually serving different purposes.

4.1.2 The TCP/IP approach

In the beginning the OSI model looked promising, many were those who thought the OSI model should exclude all other existing models. But this never happened, actually the opposite took place and the TCP/IP suite became one of the most used models. There are many reasons for the TCP/IP domination.

1. The Internet uses the TCP/IP suite. This means there is much more equipment for TCP/IP, which lowers the price of such equipment.

2. The vendors do not always have time to wait for an standardization. When the OSI model took a long time to develop, the vendors used the existing protocols, such as TCP/IP.
3. TCP/IP was developed by the U.S. military, who are big enough to control what technique the vendors shall use.

The protocol suite for the TCP/IP suit is shown in the left part of figure 2. As seen there is only 4 layers in this stack. The right part of figure 2 shows some protocols applied to each layer.

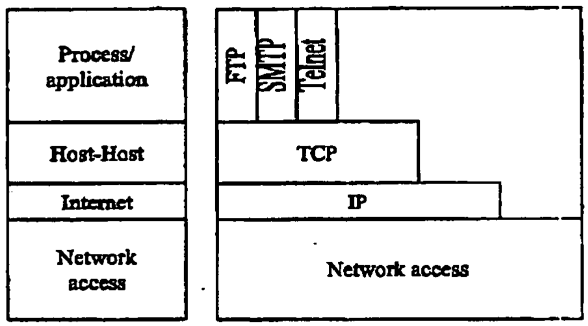


Figure 2: The TCP/IP stack to the left and some corresponding interfaces to the right.

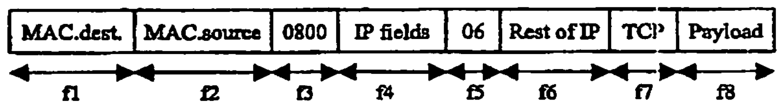
The layer in the OSI model is often referred to as layer 1 to layer 7 (L1 to L7), where L1 is the physical layer. In the TCP/IP stack there is only 4 layers but they are not called L1-L4, as one may think. The lowest level in TCP/IP reaches much higher than the lowest level in the OSI model. Therefore the lowest level is referred to as L1 and L2. The next coming layer is however numbered sequential, L3-L5. This statement is not always true, but for the text in this paper it is assumed to be.

4.1.3 Other protocol suits

There exist a large amount of various protocol suites. Many of them are now considered obsolete but some still in use are Appletalk, Novell IPX and DECnet. It is hard to predict what will happen in the future but it is safe to say that TCP/IP will grow.

4.2 The packet concept

All modern computer communication takes place with the exchange of packets. A packet contains a limited amount of data, and several packets may be needed to send the complete information. The data to be sent is encapsulated in several layers, e.g. the layers in the TCP/IP suite. The information in these layer is



f1: The receivers MAC address.
f2: The senders MAC address.
f3: The value 0x0800 tells that L3 is IP.
f4: The first fields in IP contains IP version, type of service and others.
f5: This field in the IP header tells that L4 is TCP.
f6: The rest of the IP header contains IP addresses and others.
f7: The TCP header.
f8: The payload, i. e. the information being sent.

Figure 3: A typical packet

needed to route the packet to the correct receiver and for the receiver to know what kind of information it receives.

4.3 A typical packet

This chapter is called ethernet networking, but so far ethernet has not even been mentioned. Ethernet is the lowest layer treated in this paper. It is said to be a L2 protocol, below ethernet there is the physical layer. There are several kinds of ethernet encapsulations (see appendix D) but they have many things in common. They always start with a six byte long MAC (medium access control) destination address followed by a six byte long MAC source address. Every ethernet network cards have a unique MAC address, this is needed for the network card to know which packets are addressed to itself. Except for the MAC fields every ethernet protocols have some information that specifies the type of the next coming layer. On the Internet this often is IP (Internet Protocol), in other applications IPX is a common protocol. The third layer (L3) contains, among others, routing information, which in IP is the IP destination and source addresses. There is a difference between the MAC and the IP addresses, the MAC address is used to route in local area networks (LAN) and the L3 address is used in bigger networks, so called wide area network (WAN). L3 also points out the next coming layer type, L4. At the Internet, TCP and UDP is common L4 types. A typical packet is shown in figure 3.

4.4 Switches

When several LAN:s or WAN:s shall exchange information there need to be some unit connecting the different LAN:s and WAN:s. The unit must also know how to send information between these different LAN:s and WAN:s. An intelligent type of such a unit is the switch.

Here is one example of a switch task:
A packet addressed with IP source address X arrives at one port. The switch must not onely know to which port this packet shall be sent. It also must know which source MAC address the receiver has and attach it to the packet. This knowledge can not be known by the sender since it is private to the LAN.

5 Specification of the master thesis

In a switch (see 4.4) there are several units and one of them is the decoding unit. The decoding unit is responsible for, which the name intimates, decoding the incoming packets. This means the unit put together information from the packets and gives it over to the rest of the switch e.g. which L2, L3 and L4 protocols is being used, the IP source/dest. addresses they inherits and so on.

Static packet decoders exist today but the task of this work were to produce a programmable packet decoder. A programmable decoder makes it easy to change/add protocols to the decoders tasks without fabricating a new ASIC.

In Switchcores existing switch there exist a decoder. However the design produced in this work can not directly be placed inside this since new features needs a new interface. The goal were to make the new interface look as similar to the existing as possible.

Since the design can't fit the existing product it will be constructed as if it was to be produced on a separate chip. This involves work such as Verilog coding, simulation, synthesis, timing analyzation, place & route and back annotated simulation.

If this work should be god enough to use in a future product it has to support all features performed in the static design. These are:

- Present which type of L2,L3 and L4 is being used in the current packet.
- Extract fields from the packet needed by the rest of the switch. Examples of such information is TCP source/dest. ports, ethernet type, IP-TOS field, etc.
- Perform checksum control on IP frames.
- Report incorrect packets.
- Report if the ethernet frame is of type VLAN tag or not.

The information produced in the second point is presented in something called the Option field, this is a 18 byte long register. The information from the other points is presented in the Result field, which is a three byte wide register.

The maximal area to be used was $0.5mm^2$ using a $0.25\mu m$ process and the traffic arrives to the unit at 125 MHz.

6 The design process

If an architecture is to be created, the first thing one often does is searching for published articles of the topic. This was done in this work as well, but nothing meaningful came up. So the work had to start from scratch, which mean a lot of analysis was needed. As mentioned in the abstract, a prestudy of the topic was done during the course VLSI architecture, this was a help to avoid pitfalls. The basic architecture was done with pencil and paper, but it is hard to visualize all cases that can occur. Therefore a simulator, named Calculus, to test the architecture was created.

Together with Calculus there was a need for other programs, all of them described in this chapter. All of the programs are written i C/C++. The implemented architecture was called Tintin and the created software was given names from the Tintin stories.

6.1 Calculus

A first model of the architecture was made on paper and when the simulations started the evaluation of the first model led to the final solution. Very little knowledge of the physical sizes were known in the beginning of the simulations, e.g bit length of the registers and memory sizes. Therefore, Calculus hade the possibility to change these parameters. The most crucial question was the size of the delayline (discussed in chapter 7.3.1) and its muxable part. With a short delayline Tintin would not be able to handle many different protocols and a big delayline would make area and timing constraints hard to meet. Many simulations with different sizes, new instructions and new functionality was made in Calculus. To be sure that the simulated architecture was reasonable verilog code was implemented and synthesized for a rough timing analyze. The results from the simulations are described in section Architecture.

There were one other advantage with Calculus, it was easy to study the efficiency of the program code. It is not a trivial task to write efficient code, but Calculus made it easy to trace and debug the code.

6.2 Haddock

To make the simulations meaningful (both for Tintin and Calculus) real program code was needed. In order to make programming easier a assemblator, Haddock, was developed. Haddock can handle labels, subroutines and remarks, which makes the code readable. The syntax for Haddock is described in Appendix B. Haddock made it possible to write and change the instruction code fast.

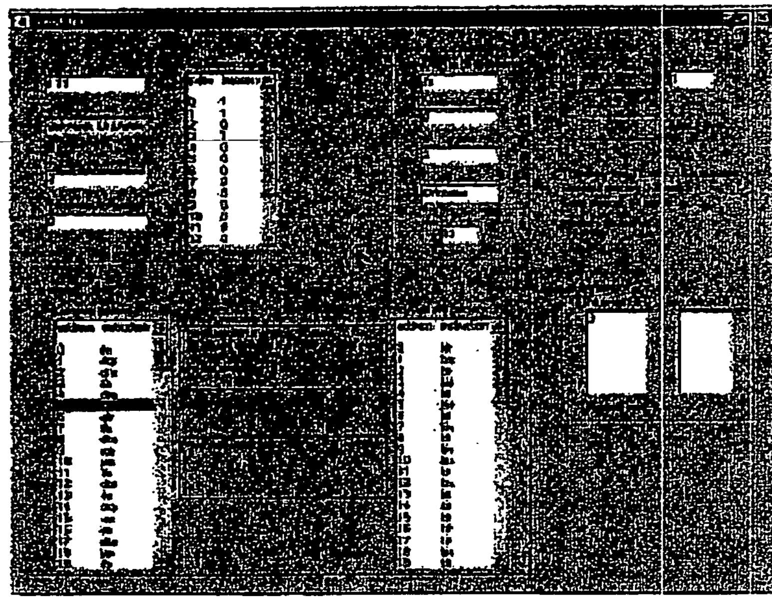


Figure 4: The simulator, Calculus

6.3 Milou

Not only the program code was needed for the simulations, also real traffic was necessary. Therefore Milou, a packet generator was written. Milou randomly generate a packet consisting of L2, L3, L4 and a payload. At layer two the following protocols can be generated:

- IEEE 803.3/802.2 SNAP
- Ethernet II
- Netware 802.3 RAW
- IEEE 802.3/802.2

At layer tree:

- IP
- IPX

At layer four:

- TCP
- UDP

other features are:

- Generate checksum error on IP headers.
- Vary the payload and the inter frame gap.
- Generate IP options field.
- Generate length errors in all layers.

There are two output files from Milou, one that contains the data in decimal form and one file describing the generated data in text.

6.4 Dupond

Both Tintin and Calculus produces a resultfield, to make sure that this field is correct when many (thousands) packets is decoded some sort of automatizend process i needed. The program Dupond was the solution for this. Dupond compares the resultfield from Calculus/Tintin and the text file from Milou. If there are errors in the result file, which ones and how many is reported. Dupond was a big help in debugging Tintin.

6.5 Implementation

When the simulations was done the architecture should be implemented in Verilog, which is a HDL language. When translating the simulated architecture some smaller problems occurred, some structures and instructions had to be changed to better fit in hardware. This made Calculus and Tintin incompatible and Calculus have not been updated since. Because no functional changes were made between Calculus ant Tintin all other programs could be used in the debuging work with Tintin.

6.6 Floor planning and Place & Route

After the synthesis, using Synopsis, there are some steps before sending the design for fabrication. This was done with Apollo and the process can be sen in picture 7.
From the final step three files were exported, a verilog file, SDF file and a GDS file. The verilog file is a netlist for the whole design, it now consists of simple gates. The SDF file contains the timing information for all gates and the connections. The GDS file can be sent to a factory for manufacturing.

6.7 Back annotated simulation

The first simulations of Tintin were made on verilog code on a very high level. When the code i synthesized errors can occur and the first simulations had no timing information. Therefore the construction was simulated again, this time using the verilog netlist and SDF files from Apollo. When this was done and no errors occurred it is safe to say that Tintin works in reality.

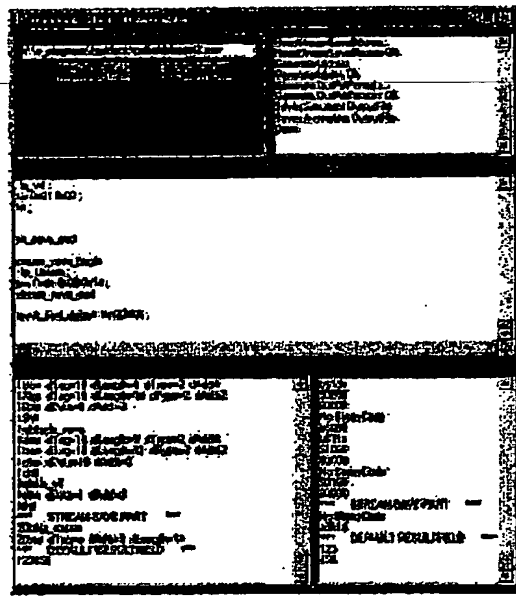


Figure 5: The compiler, Haddock

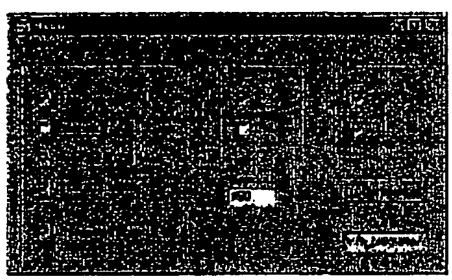


Figure 6: The generator, Milou

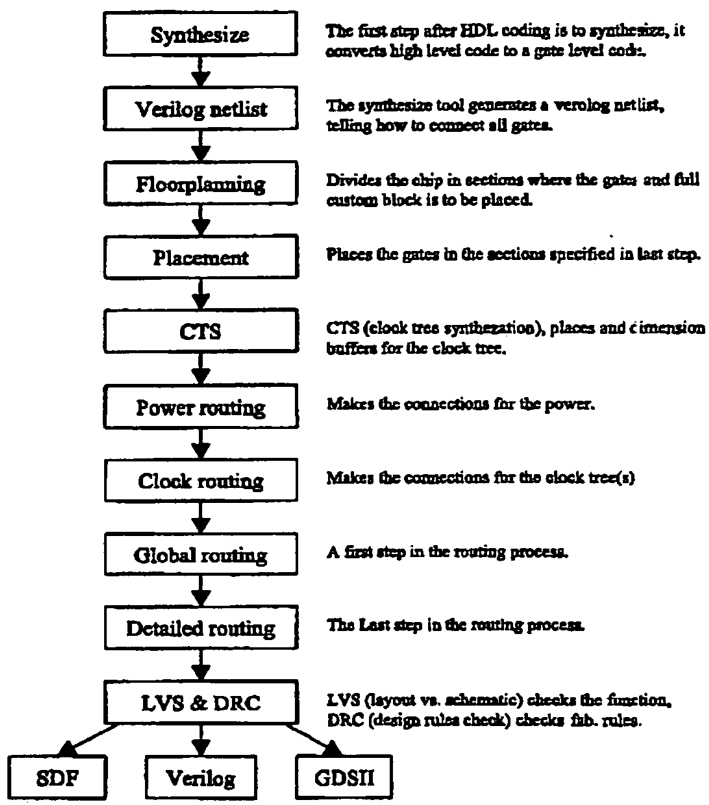


Figure 7: Designflow

7 The Tintin architecture

7.1 The interface

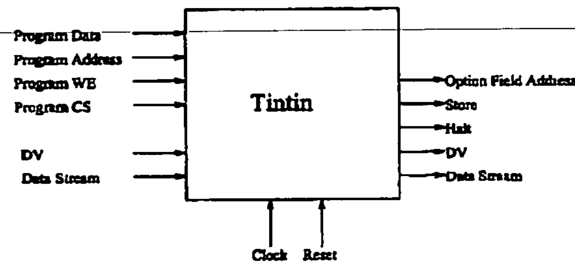


Figure 8: IO-interface

The input interface to Tintin is simple, there is one synchronous serial byte stream for the data that should be decoded together with a 1-bit data valid signal(DV) and also some used to program Tintin after power on.
The output interface consists of a serial byte stream together with some control signals. The control signals are a data valid(1 bit), an option field address(6 bits), a store and a halt signal(1 bit each). The store signal tells if the current byte is to be stored in the option field, the halt signal together with the store signal tells if the stream out is the inserted result field. The address bus allows addressing in the option field.
The programming interface consist of an 8-bit address bus, a 18-bit data bus, a chip select and a write enable signal for programming the two instruction memories and some specific registers described later in the text.

7.2 How does it work

When the data stream enters Tintin, it is passed through a *delayline*, see fig.9. This is a 23 shift deep and 1 byte wide shiftregister. As long as a byte is in the first 16 positions it can be accessed by the *compare processor*, basically a parser. The *compare processor* is responsible for decoding the packets, it is connected to a instruction memory which inherits the parsing code.
One basic property in the Tintin architecture is that every incoming byte is numbered with a tag. When the *compare processor* asks for a specific tag the *mux control* delivers the byte on that position.
When the parser have come to some kind of conclusion it might want to report something to the result field or the option field, see chapter 5. This is done by starting up a save sequence. A start address for a save sequence will be sent from the *compare processor* to the *save engine*. The *Save engine* examine the incoming address and decides if it is a save regarding the result field or the option field. According to this decision the address is placed in either the *bit save fifo* or the *stream save fifo* respectively.
The bit save unit has three functions, it can set bits in the result field, perform

Tintin

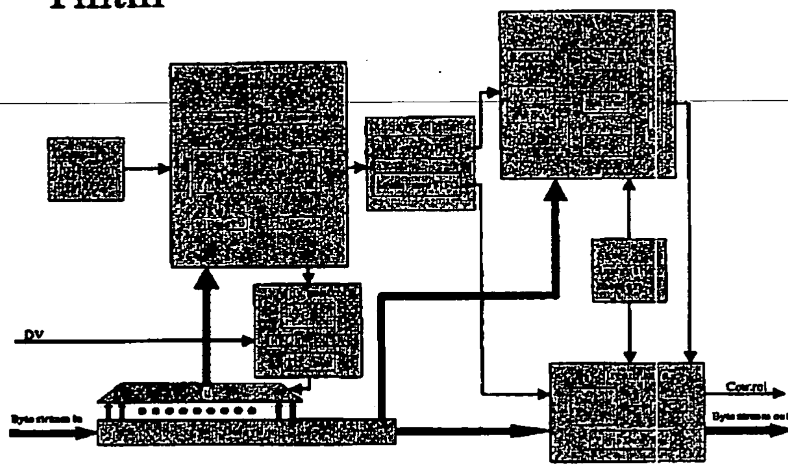


Figure 9: Block diagram of Tintin

checksumcontrol and lengthcontrol. The stream save unit execute the instruction that saves the option field.
The stream save unit also inserts the result field in the stream and regulates the control signals.

7.3 Detailed description

7.3.1 The Delayline

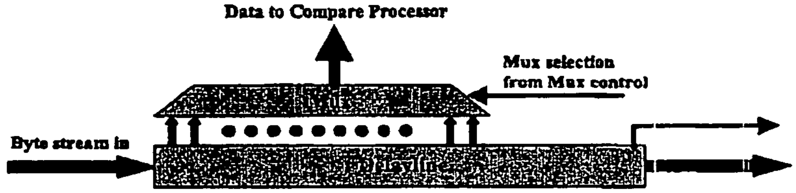


Figure 10: The Delayline with mux

The *delayline* is rather simple, it consists of a 23 shift deep, 1 byte wide shiftregister. The 16 first positions of the shiftregister is reachable from the *compare processor* through a mux. The two last positions are connected to the save blocks(*bit save* and *stream save*). The stream save block is actually only using the very last position, only the bit save block needs the last two positions because the checksumcontrol works with 16 bits at a time. There are

five positions that can not be accessed either by the parser or the saveblocks(*bit save* and *stream save*). The reason for this delay before the byte stream arrives to the save blocks is that all start addresses sent from the compare processor to the saveblocks are cued in a fifo. In some extreme situations when many these five shifts may not be enough, which will generate an error. The actual delay to be sure that an error never will occur is 4*64=192 clock cycles, 64 is the maximum length of a save sequence and 4 is the maximum of start addresses waiting to be executed in one of the saveblocks, but the simulator, Calculus, have showed that five delay cycles is enough.

7.3.2 Muxcontrol

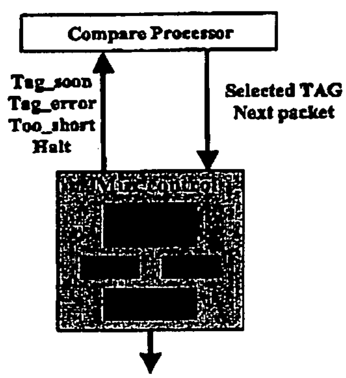


Figure 11: The Mux controller

A fundamental function for Tintin is that it automatically keeps track of where a specific byte has its location in the delayline. The programmer only need to specify which tag, i.e. which number the byte has, where the first byte in a packet is number zero, the second is number 1 and so on. This is why every byte arriving to the *delayline* is tagged (numbered). This could be done easily by just adding an extra field in every shift in the *delayline* inheriting the bytes tag. This is bad in two points of view, first much silicon would be used to implement the extra field in the *delayline*. Second, when the parser wants to look at a specific tag it would take a lot of time if every shift should be searched to find the wanted tag. This is why an alternative implementation is used.
The muxable part of the *delayline* is the 16 latest incoming bytes. Worst case for a packets length is 1 byte, but since the first 12 bytes always contain the MAC-address, no useful information can be extracted if the packet is shorter than 13 bytes. These packets will force the *compare processor* to begin with the next packet at once and their DV signal will be unset so the rest of the switch will never see it. With a limit of at least two clock cycles(bytes) between different packets it is possible to guarantee that never more than two packets

exist at the same time in the *delayline*. According to the ethernet standard the IFG(Inter Frame Gap), which means the distance between packets, are at least 20 cycles, but a smaller distance is always desirable. E.g. with minimum distance of 6 cycles makes it possible to easy extend Tintin to be able to take care of SONET frames.
The *mux controller* uses two identical *TU* (TagUnits), one for each possible packet, a *CS* (Controlling Statemachine) to control the *TU:s* and a *TU mux* to choose which one of the *TU:s* that the *compare processor* is interested of.
A *TU* consists of two registers (tagfield and the lastfield), some adders and a rather simple statemachine. When a packet arrives, the tagfield starts to increment for every byte. When the DV signal becomes false again the tagfield stops counting and the lastfield starts to increment. The *TU* sends an 'end_of_packet' signal when the lastfield reaches the number of shifts in the delayline. If the packet was shorter than 13 bytes a 'too_short' signal will be generated.
The position of a requested byte is located according to

$$\mathbf{p = tagfield + lastfield - wanted\_tag}$$

The *TU* also generates a 'tag_error' if the requested tag never will be available or 'tag_soon' if the requested tag has not arrived to the *delayline* yet.
The *CS* is responsible for selecting a free *TU* for an arriving packet and and to pause the *compare processor* when no new packets are available. The *CS* will unselect a *TU* when the *TU* generates an 'end_of_packet'. An unselected *TU* will be reset to prepare it to receive the next incoming packet. The *CS* is also controlling the *TU mux* to change its state every time the *compare processor* is asking for a new packet.

7.3.3 The Compare processor

The brain of Tintin is a programmable parser, see instructionset in appendix A. It uses four registers to fulfill it tasks.

- One PC register that hold the value of the program counter
- One general register. It can be used with the instructions *otr* and *ifr*.
- One base register, called basereg. When the parser searches a tag, the value in the basereg is added to the searched tag value. This is used to be able to reuse instructioncode for e.g. L3 frames, even if they are encapsulated in different L2 frames. The instructions *otr* and *atm* can change this register.
- One stack address register used to store addresses when subroutine instruction is called with *jsr*. It is only possible to store one address in the stack. *rtn* copy the stack back to the PC.

All instructions are executed in one clockcycle, except in two cases. First when the received instruction is of type *ifs* or *ifm*, then it compares the first data in that block in the same cycle. Second when the next instruction is of type *jas*, *jsr* or *rtn*, then the PC is updated at once. This is possible because the *compare processor* unit receives two instructions every clockcycle from a double ported memory. These two features decreases the total amount of clock cycles

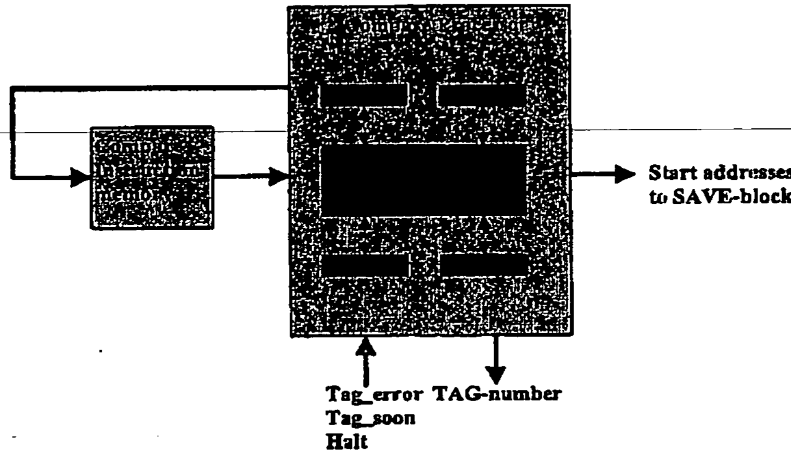


Figure 12: The Compare Processor

needed for the *compare processor* to parse a packet, thereby decreasing the size of the *delayline*. The *ifs* and the *jas* instructions are able to start save sequences, these instructions have a field that tells what address in the save memory that shall start the execution. Save address 0x00 will not generate a start of a save sequence.
The *compare processor* must know when a new parsing is started so the registers can be reset. Therefore, when parsing of a packet is done, there shall be a *jas* instruction with jump address 0x7f (=last compare instruction memory address). When this is detected it resets and starts looking for a new packet. If the *compare processor* gets the signal 'too_short' it is reset. 'tag_soon' pauses the processor and 'tag_error' force it to begin with the next packet.

7.3.4 Save engine

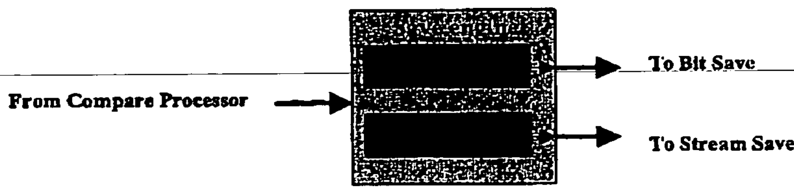


Figure 13: The Save Engine

The save engine takes the address sent from the compare processor and determine if it is the start address of a bit save sequence or a byte stream sequence. After this the address together with the current BASE is put in the specific fifo. The BASE is needed for all save instructions that is using tagnumbers. When Tintin is programmed, a constant is written to the save engine to tell where bit save sequences ends in the save memory. This feature exist because it is hard to tell how many instructions are needed to the the different parts and it is more expensive to map two memories than one twice as big.

7.3.5 Bit save

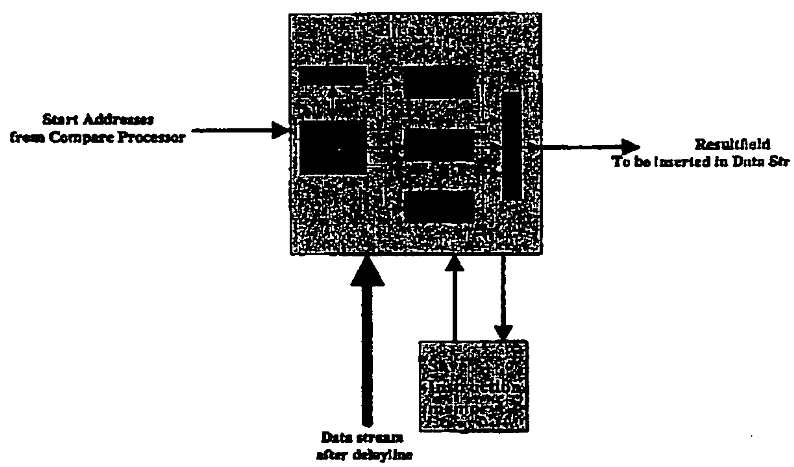


Figure 14: The Bit Save processor

The bit save unit writes to the result field. The result field consist of 24 bits

or 3 bytes. It is controlled by the save instruction memory and orders other units to execute the instructions. The executing units are:

- **Checksum**, this block executes the *csc* instruction which performs a 16-bit one complement addition. The unit needs to know what tag to start the execution from(Tag) and how many bytes the checksum should cover(Length). If there are checksum errors (i.e the sum differs from 0xFFFF) the unit writes to the result field. Which bit in the result field to write to is static and can not be changed by the programmer.
  Further this block need the value of the BASE as it was when the compare processor sent the start address of the current save sequence.

17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

17–16	15–8	7–0
10	Tag	Length

- **Bit,** This unit executes the *bis* commands which bitwise 'xor' one selected byte in the result field with the data field. In other words, all bits which are set in the data field will invert the corresponding bit in the result field. It is only possible to invert one specific bit one time per packet, this is because e.g. a L3 error could be found in many ways, but if the the bit which indicate L3 error is set an even number of times, this would look like a correct L3 packet in the result field. The address field tells to which one of the total three bytes in the result field to write to.

17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

17–16	15–8	7–6	5–0
01	Data	Addr	XXXXXX

- **Length error,** this is the most complex unit and investigates lengths in a packet and is used with the *len* instruction. In a network there might occur packets that has been cut of. This causes many sorts of errors, e.g. if layer 4 is shorter than two bytes the result field should indicate L3 error but not L2 error.
  The unit consists of two identical checkboxes and one controller. A checkbox needs to know at which tag to start the measurement from, what kind of comparison it is supposed to perform (more,less or equal) and what length to match this comparison to. If a checkbox detects a length error, the field Addr tells to which one of four possible bits in the results field to write to. As with the checksum unit, this unit also needs the value from the BASE as it was when the compare processor sent the start address of the current save sequence.

17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

17–16	15–10	9–4	3–2	1–0
11	Tag	Length	Type	Addr

7.3.6 Stream Save

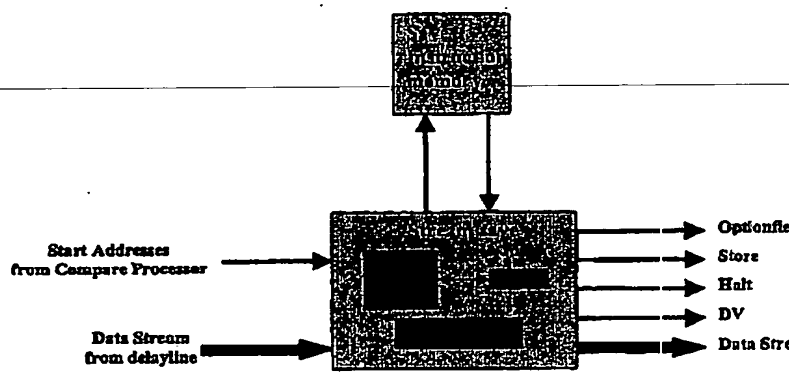


Figure 15: The Stream Save processor

Stream save has only one save instruction to handle, the *bss*. It is used to save to the option field and includes a start tag number, a length and an six bit wide address to tell were in the option field the selected bytes are to be written. Besides of this it also inserts the result field as soon as all bit save instructions are executed.

17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

1	Tag	Address	Length

7.4 Programming Tintin

The programming of Tintin is done trough the inputs PROGRAM DATA, PROGRAM ADDRESS, PROGRAM WE and PROGRAM CS, see fig.9. Tintin will keep DV OUT low while CS is active. The memory mapping inside Tintin is shown in fig.16.

On address 0xF0 the default result field is programmed. This is needed since some bits in the result field might indicate things that can't be detected, like no existing L4. This bit must instead get cleared when a legal L4 is found.

The last register is needed for the Save Engine to decide weather an instruction is to be sent to the Bit Save or the Stream Save unit.

0x00 … 0x7F	Compare Instruction memory
0x80 … 0xBF	Save Instruction memory
0xC0	Number of Bit Save instructions
0xC1 0xC2	Default resultfield (Only the 12 LSB are used)
0xC3 … 0xFF	Not used

Figure 16: Memory map

8 Results , limitations and further development

The constrains from chapter 5 have been fulfilled. The area for the total implementation, memory as well as logic, was $0,49mm^2$ and can be clocked at 140 MHz. These figures are god estimations of the real values since they are produced on a low level, much more exact than figures from synthesis tools.

During the work considerations to support ISL and SONET has been taken. They can not be direct supported but very little extra logic is needed, specially for SONET.

Lgg till mer om Limmintations och fortsatt utveckling.

9 References

References

[1] Cisco Systems, Inc. *Cisco IOS Solutions for Network Protocols, Volume II: IPX, Apletalk, and more*, Macmillian Technical Publishing 1998

[2] IEEE *Various IEEE-documents(RFC) concerning internet protocols*

A Instructionset

A.1 Compare Processor

IFM, IF stream with Mask

	17 16 15	14 13	12 11 10 9	8	7 6 5 4 3 2 1 0
Addr: PC	000	Cmp	If length	X	Mask
Addr: PC+1	Tag			Data	

IF (Condition True) then PC = PC + 2
ELSE PC = PC + Iflength + 2

Cmp: Equal = 00
LessThan = 01
MoreThan = 10
NotEqual = 11

Condition is True if Data relative to the masked streamdata, fulfill the Cmp specified.

IFS, IF stream with several bytes and Save

	17 16 15	14 13	12 11 10 9	8 7	6 5 4 3 2 1 0
Addr: PC	001	Cmp	If length	DL	Save Address
Addr: PC+1	Tag1			Data1	
⋮	⋮			⋮	
Addr: PC+N	TagN			DataN	

IF (Condition True) then PC = PC + DL* + 1
ELSE PC = PC + DL* + Iflength + 1

Cmp: Equal = 00
LessThan = 01
MoreThan = 10
NotEqual = 11

Condition is True if Data relative to streamdata fulfill the Cmp specified.
If Save Address differ from zero and IF (Condition True),
then the save Address will be sent to the Save Processors.

Data Length

IFR, IF Reg

	17 16 15	14 13	12 11 10 9	8	7 6 5 4 3 2 1 0
Addr: PC	010	XX	If length	X	Mask

IF ((Reg & Mask)==0) then PC = PC + 1
ELSE PC = PC + Iflength

OTR, Operate To Reg

	17 16 15	14 13	12	11	10 9 8	7 6 5 4 3 2 1 0
Addr: PC	011	Src	Dst	X	Operation	Value

Src: CmpBase = 0 , Source = BASE
Reg = 1 , Source = REG
Tag = 2 , Source = (Value)
Konstant = 3 , Source = Value

Dst: CmpBase = 0 , Destination = BASE
Reg = 1 , Destination = REG

Operation: Add= 0 , Destination = Destination + Source
Sub = 1 , Destination = Destination - Source
Write = 2Destination = Source
Shift = 3 Destination = Destination ¡¡ Source
And = 4, Destination = Source & Destination
Or = 5 , Destination = Source — Destination

Note: Value is not always used

ATM, Add Tag with Mask

	17 16 15	14 13 12 11 10 9 8	7 6 5 4 3 2 1 0
Addr: PC	100	Tag	Mask

Masks and adds streamdata with tagnumber Tag to BASE.

JAS, Jump And Save

	17 16 15 14	13 12 11 10 9 8 7	6 5 4 3 2 1 0
Addr: PC	1010	Save Address	Jump Address

PC = jump Address

JSR, Jump and save to SubRoutine

	17–14	13–7	6–0
Addr: PC	1011	Save Address	Jump Address

STACK = PC
PC = Jump Address

RTN, ReTurN from subroutine

	17–15	14–0
Addr: PC	110	XXXXXXXXXXXXXXX

A.2 BitSave Processor

BIS, BIt Save

17 16	15 14 13 12 11 10 9 8	7 6	5 4 3 2 1 0
01	Data	Addr	XXXXXX

CSC, CheckSum Control

17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0
10	Tag	Length

LEN, LENgth control

17 16	15 14 13 12 11 10 9	8 7 6 5 4	3 2	1 0
11	Tag	Length	Type	Addr

HLT, HaLT

17	16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0	XXXXXXXXXXXXXXXXX

Note: All save sequences ends with a halt instruction.

A.3 StreamSave Processor

BSS, Byte Stream Save

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	Tag							Address						Length			

HLT, HaLT

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	XXXXXXXXXXXXXXXXX																

Note: All save sequences ends with a halt instruction.

B Assembler syntax

C A program example

Here is a programm fore Tintin that solves the same tasks as the static design of today does.

```
{
# ********** Main L2 ************************
    : VLAN? ;
    ifs eq 0x0c,0x81 0x0d,0x00 ; save tag_save ;
    # to find out if the packet is vlan tagged.
    {
        ifm eq 0x0e,0x10 ; mask 0x10 ;
        {   otr const reg1 write 0x01 ;
                # to remember that the packet has a RIF field
        }
        otr const cmpbase write 0x04 ;
            # compensate for EDIF and TCI fields
    }

    ifs lt 0x0c,0x06 ;
        # find out witch L2 it is
    {
        jas dummy_save NotEthII ;
            # It is not EthernetII
    }
    jas EII_save detect_L3 ;
        # it is EthernetII

# ********** Main L3 ************************
    : detect_L3 ;
        # Start detection of L3
    ifs eq 0x0c,0x08 ;
        # separate between ARP,RARP,IPv4,IPv6 and IPX
    {
        ifs eq 0x0d,0x06 ;
        {
            jas dummy_save ARP_ex ;
                # ARP detected
        }
        ifs eq 0x0d,0x35 ;
        {
            jas dummy_save RARP_ex ;
                # RARP detected
        }
        ifs eq 0x0d,0x00 ;
        {
            jas dummy_save IP_ex ;
                # IPv4 detected
        }
        jas dummy_save end ;
```

```
            #If the execution reaches this
            #point it is a unknown 13.
    }
    ifs eq 0x0c,0x86 0x0d,0xdd ; save IP_save ;
    {
        jas dummy_save IPv6_ex ;
            # IPv6 detcted
    }
    ifs eq 0x0c,0x81 0x0d,0x37 ; save IPX_save ;
    {
        jas dummy_save IPX_ex ;
            # IPX detected
    }
    jas dummy_save end ;
            #If the execution reaches this
            #point it is a unknown 13.

# **************** Not EII *****************
    : NotEthII ;
    jsr dummy_save RIFF_length ;
        # subrutin call to take care of the RIF length.
    ifs eq 0x0e,0xe0 0x0f,0xe0 ; save noSNAP_save ;
    {
        # It is 802.3/802.2 with IPX
        ifs eq 0x10,0x03 ; # mask 0x03 ;
            # This ifs-block finds out how long the LLc part is.
        {
            otr const cmpbase add 0x03 ;
                # compensate for the LLC field, 3 detected.
            jas IPX_save IPX_ex ;
        }
        otr const cmpbase write 0x04 ;
            # compensate for the LLC field, 4 detected.
        jas IPX_save IPX_ex ;
    }
    ifs eq 0x0e,0xaa 0x0f,0xaa 0x10,0x03 ; save SNAP_save ;
    {
            # it is SNAP.
        otr const cmpbase add 0x08 ;
            #compensate for the length,
            #idstring(AAAA03) and the org. code.
        jas dummy_save detect_13 ;
    }
    ifs eq 0x0e,0xff 0x0f,0xff ; save RAW_save ;
    {
            # netware raw with IPX.
        jas IPX_save IPX_ex ;
    }
    jas dummy_save end ;
        #the programm should newer reach this point,
```

C A PROGRAM EXAMPLE

```
          #if so it is a unknown 12

# ********** IPX extraction *****************

     : IPX_ex ;
          #No IPX parsing is done.
     jas dummy_save end ;

# ********** IPv6 extraction *****************

     : IPv6_ex ;
          #No IPv6 parsing is done.
     jas dummy_save end ;

# ********** ARP extraction ********

     : ARP_ex ;
          #No ARP parsing is done.
     jas dummy_save end ;

# ********** RARP extraction ********

     : RARP_ex ;
          #No RARP parsing is done.
     jas dummy_save end ;

# ********** IP extraction ********

     : IP_ex ;
     jar dummy_save RIFF_length ;

     jas ip_save nrow ;
     : nrow ;
     jas IP_stream nrow2 ;
     : nrow2 ;

     ifm eq 0x0e,0x40 ; mask 0x40 ;
     {
          jas is_v4 dumlabel1 ;
          : dumlabel1 ;
     }

     : l4 ;
     ifs eq 0x17,0x01 ; save OTHERl4_save ;
     {
          jas dummy_save the_end ; #L4 is ICMP
     }
     ifs eq 0x17,0x02 ; save OTHERl4_save ;
     {
          jas dummy_save the_end ; #L4 is IGMP
```

```
    }
    ifs eq 0x17,0x06 ; save TCP_save ;
    {
        jas dummy_save the_end ; #L4 is TCP
    }
    ifs eq 0x17,0x11 ; save UDP_save ;
    {
        jas dummy_save the_end ; #L4 is UDP
    }
    jas OTHERl4_save the_end ;
    : the_end ;
    jas dummy_save end ;
# ************ subroutin RIFF-part ****************
    : RIFF_length ;
    ifr ; mask 0x01 ; # RIFF ?
    {
        rtn ;
            #no rif part, return from subroutin.
    }
    ifm eq 0x0e,0x40 ; mask 0xc0 ;
        #Find out if there is an RD field
    {
        otr const cmpbase add 0x02 ;
            #RD field not precent, compensate for the RD field.
        jas dummy_save back ;
    }
    atm 0x0e 0x1f ;
        #RD field is precent, reads RIFF lengt from stream.
    : back ;
    otr const reg1 write 0x00 ;
        #Remove tag marker from register.
    rtn ;
}


bit_save_begin

: dummy_save ;
hlt ;

: tag_save ;
bis 0x02 0x01 ;
hlt ;

: EII_save ;
bis 0x04 0x01 ;
hlt ;

: SNAP_save ;
bis 0x08 0x01 ;
```

C A PROGRAM EXAMPLE

```
hlt ;

: noSNAP_save ;
bis  0x10 0x01 ;
hlt ;

: RAW_save ;
bis  0x0c 0x01 ;
hlt ;

: IP_save ;
bis 0x01 0x01 ;
bis  0x20 0x01 ;
csc 0x0e 0x14 ;
hlt ;

: IPX_save ;
len 0x0e 0x1d mt 0x01 ;
bis 0x01 0x01 ;
bis  0x40 0x01 ;
bis 0x20 0x02 ;
hlt ;


: OTHER14_save ;
bis 0x18 0x02 ;
hlt ;

: TCP_save ;
len 0x18 0x09 mt 0x01 ;
len 0x18 0x1d mt 0x02 ;
bis 0x08 0x02 ;
hlt ;

: UDP_save ;
len 0x18 0x09 mt 0x01 ;
len 0x18 0x11 mt 0x02 ;
bis 0x10 0x02 ;
hlt ;

: is_v4 ;
bis 0x01 0x02 ;
hlt ;


bit_save_end

stream_save_begin :
IP_stream ;
bss 0x0e 0x00 0x14 ;
```

```
stream_save_end

result_field_default 0x123456 ;
```

D Common packets

Untagged Ethernet frames

IEEE 802.3/802.2 SNAP Encapsulation (RFC 1042)

DA	SA	Length	AA AA 03	Org. code	EType	Data	CRC
6	6	2	3	3	2		4

Ethernet II Encapsulation (RFC 894)

DA	SA	Etype	Data	CRC
6	6	2		4

NetWare 802.3 RAW Encapsulation (IPX only)

DA	SA	Length	IPX-Checksum	Data (without IPX checksum)	CRC
6	6	2	2		4

IEEE 802.3/802.2 Encapsulation

DA	SA	Length	LLC	Data	CRC
6	6	2	3 or 4		4

DA Destination MAC Address
SA Source MAC Address
Length Length of packet (excluding DA, SA, Length and CRC) (must be less than 0600 $_{16}$)
EType Ethernet Type of data (always >=0600 $_{16}$)
0800_{16} = IP datagram
0806_{16} = ARP
0835_{16} = RARP
8137_{16} = IPX datagram
$86DD_{16}$ = IPv6
Org. code A three byte organization code. All codes are allowed.
IPX-Checksum The Checksum of the IPX frame.
LLC The Logical link control.
E0 E0 XX_{16} (XX_{16}) = IPX.

LLC

DSAP	SSAP	Control
1	1	1 or 2

DSAP Destination Service Access Point
SSAP Source Service Access Point
Control Control fields (see below)

Control

Byte 1		Byte 2 (only present in I- and S-format PDUs)
N/A	Type	N/A
7 6 5 4 3 2	1 0	7 6 5 4 3 2 1 0

Type $X0_2$ = I-format PDU (2 bytes Control field)
01_2 = S-format PDU (2 bytes Control field)
11_2 = U-format PDU (1-byte Control field)

Data Type of data
CRC Cyclic Redundancy C heck of the entire ethernet frame

D COMMON PACKETS

Tagged Ethernet frames

VLAN-tagged IEEE 802.3/802.2 SNAP Encapsulation (RFC 1042)

DA	SA	ETPID	TCI	Length	RIF	AA AA 03	Org. code	EType	Data	CRC
6	6	2	2	2	0-30	3	3	2		4

VLAN-tagged Ethernet Encapsulation (RFC 894)

DA	SA	ETPID	TCI	Etype	RIF	Data	CRC
6	6	2	2	2	0-30		4

VLAN-tagged NetWare 802.3 RAW Encapsulation (IPX only)

DA	SA	ETPID	TCI	Length	RIF	IPX-Checksum	Data (without IPX checksum)	CRC
6	6	2	2	2	0-30	2		4

VLAN-tagged IEEE 802.3/802.2 Encapsulation

DA	SA	ETPID	TCI	Length	RIF	LLC	Data	CRC
6	6	2	2	2	0-30	3 or 4		4

ETPID 8100_{16}

TCI Tag Control Information

bits 16-14 *User priority* 7-0 (0 is lowest priority)

bit 13 *CFI* (Canonical Format Indicator)
set = RIF field is present in the Tag Header
reset = No RIF field is present

bits 12-1 *VID* (VLAN identifier)
000_{16} = No VLAN identifier is present (only User priority)
001_{16} = Default
FFF_{16} = Reserved

RIF The RIF field is present if CFI is set. RIF is used in a Token Ring network to provide source routing. The RIF field looks like,

RIF

RC	RD
2	0-28

RC Route Control

bits 16-14 *RT* (Routing type)

bits 13-9 *LTH* Length in bytes of whole RIF field (including RC and RD)
(valid lengths; if RT=01X $_2$ then LTH is set to 0 (zero), else 2-30

bit 8 *D*, *Direction bit*

bits 7-2 *LF*, Largest frame

bit 1 *NCFI* (Non-canonical format indication)

RD Route Descriptors (two bytes each)

D COMMON PACKETS

Ethernet Data format

There are a number of Ethernet data formats depending on the EType field (in case of EthernetII and IEEE 802.3/802.2 SNAP encoded packets) in the Ethernet header;

1. EType = 0800 ↵ IP Datagram
2. EType = 0806 ↵ ARP request/reply
3. EType = 8035 ↵ RARP request/reply
4. EType = 8137 ↵ IPX Datagram

IP Datagram (RFC791)

IP Datagram

0–3	4–7	8–15	16–18	19–31
[illegible]	IHL	[illegible]	Total length	
Identification			Flags	Fragment offset
[illegible]				
[illegible]				
[illegible]				
Options				
Data				

Version	IP datagram version
IHL	Internet Header length in 32 bits words (points to beginning of data)
TOS	Type of service - Quality of service. It is confusing that a field in the Type Of Service byte has the same name.
Total length	Length of the datagram in octets (header and data)
Identification	Used for fragmentation
Flags	Used for fragmentation
Fragment offset	Used for fragmentation
Time to live	The maximum time the datagram is allowed to remain in the Internet system (decremented by at least one by every module that process the datagram). If zero, the packet must be discarded
Protocol	Next level of protocol (defined in RFC790) 1 = ICMP 2 = IGMP 6 = TCP
Header checksum	Check sum of header only (16 bits one's compliment of the one's compliment sum of all 16 bit words in the header).
Source IP Address	The source IP address
Destination IP Address	The destination IP address
Options	May appear or not in datagrams

D COMMON PACKETS

IP Data format

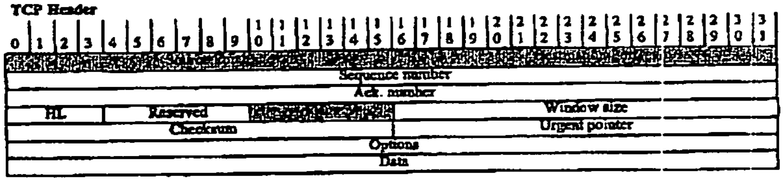
There are a number of IP data formats depending on the IP-Protocol field in the IP header

1. IP-Protocol = 01 ICMP
2. IP-Protocol = 06 TCP

IP-Protocol = 11 UDP.

TCP header

TCP Header



Source Port The TCP source port
Destination Port The TCP destination port
Sequence number Identifies the first byte in the stream of data from sending TCP to receiving TCP.
Ack. number Identifies the next sequence numbethet the sender of theacknoledgment expects to receive.
HL The length of the header in 32bit words.
Flags Six flag bits. One or more can be turned on at the same time.

Flags

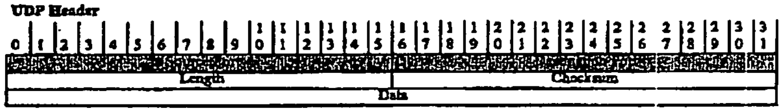
5	4	3	2	1	0
URG	ACK	PSH	RST	SYN	FIN

URG: The urgent pointer is valid.
ACK: The Acknowledge number is valid.
PSH: The receiver should pass this data to the applicati asap.
RST: Reset the connection.
SYN: Synchronize sequence numbers tinitialize a connection.
FIN: The sender is finished sending data.

Checksum Covers the TCP header and data
Urgent pointer Has to do with the sequence number.

UDP header

UDP Header



Source Port The UDP source port
Destination Port The UDP destination port
Length The length of the UDP header and data in bytes.
Checksum Covers the UDP header and data.

D COMMON PACKETS

IPX Datagram

IPX Datagram

0 – 15	16 – 31
IPX Checksum	Total length
Hop Count / Packet type	Destination Network Address (part I)
Destination Network Address (part II)	Destination Node Address (part I)
Destination Node Address (part II)	
Destination Socket Number	Source Network Address (part I)
Source Network Address (part II)	Source Node Address (part I)
Source Node Address (part II)	
[illegible]	

IPX Checksum IPX datagram checksum.
Total Length Indicate the length of the IPX header (30 bytes) plus the valid data.
Hop Count The Hop Count field (also referred as Transport Control field) indicates the number of routers or routing processes the packets has crossed. Maximum hop count differs depending whether RIP or NLSP is used.
Packet Type The Packet type field was intended to define the type of communication (SPX, NCP, SAP, RIP and so on). This field, however, has really been messed up over the years. The only values that are reliable are:
5 = SPX
20 = NetBIOS
The best way to identify a packet is by the source and destination socket numbers.

Destination Network Address
Indicates the final destination network for the packet.

Destination Node Address
Indicates the node address (or host ID number) of the destination device.

Destination Socket Number
Socket numbers indicate the end process that the packet is destined for or the source process that created the packet. Some common socket numbers:
0452_h = SAP
0453_h = RIP
0455_h = NetBIOS
9001_h = NLSP

Source Network Address
Indicates the network that the packet is being sent from.

Source Node Address
Indicates the node address (or host ID number) of the sending station.

Source Socket Number
See description of Destination Socket Number.

E Time schedule